Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

The h-index Metric for GitHub

Bachelor Thesis

2026
Mykhailo Pavlov

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

The h-index Metric for GitHub

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	Mgr. Marek Šuppa

Bratislava, 2026
Mykhailo Pavlov

Acknowledgments: Tu môžete poďakovať školiteľovi, prípadne ďalším osobám, ktoré vám s prácou nejako pomohli, poradili, poskytli dáta a podobne.

Abstrakt

Slovenský abstrakt v rozsahu 100-500 slov, jeden odstavec. Abstrakt stručne sumarizuje výsledky práce. Mal by byť pochopiteľný pre bežného informatika. Nemal by teda využívať skratky, termíny alebo označenie zavedené v práci, okrem tých, ktoré sú všeobecne známe.

Kľúčové slová: jedno, druhé, tretie (prípadne štvrté, piate)

Abstract

Abstract in the English language (translation of the abstract in the Slovak language).

Keywords:

Contents

List of Figures

List of Tables

Introduction

Motivation

The motivation for adapting the h-index for GitHub was that, unlike in the scientific world, there is no metric on GitHub that can be used to assess an individual developer's personal contribution to creating open-source projects. With the help of the h-index adaptation, by looking at a developer, one can understand that if his h-index is about 10 or higher, then he has created projects that have been appreciated.

Problem Statement

The goal is to create a new metric for evaluating the influence of users on GitHub. Currently, GitHub hosts millions of developers, but there is no metric to assess their contribution and influence in open-source. Existing options, such as total stars, total repositories, and follower counts, do not reflect the actual influence of a developer. A developer with 1,000 repositories that are not used by others has contributed less than a developer with 10 repositories, each with at least 10 stars. This work explores the h-index as a new metric for GitHub to evaluate the metric and its related features.

Research Questions

The goal of the work was to find out how the h-index is distributed on GitHub. Do GitHub followers correlate with the h-index, and does having fewer followers lead to a higher h-index? Do repositories effectively show a developer's influence in open-source? Which GitHub features are most closely related to the h-index? Can we identify what separates high-impact developers from low-impact developers?

Chapter 1

Metrics and evaluation methods

In this chapter, we introduce the key metrics and conceptual tools used to evaluate user impact within the GitHub platform.

1.1 The h-index metric

The h-index is a metric for evaluating the impact of a scientific author, originally introduced by physicist Jorge E. Hirsch in 2005 [7].

The h-index has proven its value in the academic field [8]. It has been used for over 20 years. It shows that an author does not just have a single paper that became very popular, but has a number of works that have each received a significant number of citations.

This makes the h-index an important metric for evaluating academic work. As GitHub increasingly serves as a platform for open science and research code distribution [9, 4], and as scholarly publications increasingly reference GitHub repositories [3], adapting bibliometric measures to this context is reasonable. Therefore, it was considered appropriate to apply a similar concept to GitHub as well, treating repositories as publications and stars as citations.

1.1.1 Existing definition

The h-index is calculated as the largest number h such that the author has at least h articles, each of which has been cited at least h times.

$$h = \max\{\, i \in \{1, \ldots, n\} : c_i \geq i \}\tag{1.1}$$

1.1.2 Adapting the h-Index to GitHub

The GitHub h-index will operate by using stars and repositories instead of citations and articles, respectively. With these basic elements, it becomes possible to measure the impact of the user.

$$h_G = \max\{i \in \{1, \ldots, n\} : s_i \geq i\} \tag{1.2}$$

This means that the user has at least h_G repositories, each with at least h_G stars.

Example of GitHub h-index calculation

We are given an array containing the number of stars for each repository. First, we sort this array in non-increasing order. Then, we iterate through the array while maintaining a variable i, initially set to 0. On each iteration, we check whether the current number of stars is greater than or equal to i. If so, we increment i and continue. The process stops when the number of stars is less than i. At this point, the user's h-index is $i - 1$. The result of the calculation is illustrated in Figure 1.1.

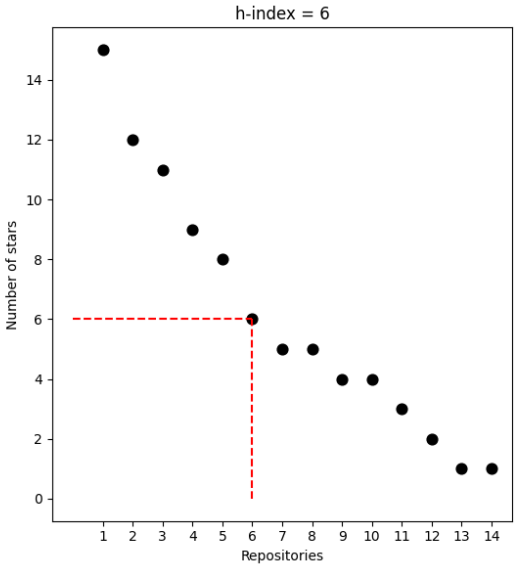

Figure 1.1: A visual example of a user's h-index calculated based on their repositories and the number of stars.

1.1.3 Realization of the GitHub h-index calculation

The realization of the GitHub h-index is based on the definition introduced in Equation 1.2. The input of the algorithm is a list of integers, where each value represents the number of stars assigned to a single repository owned by a given GitHub user.

As a first step, the algorithm validates the input data. An empty list indicates that the user has no repositories, in which case the resulting h-index is equal to zero. Additionally, the algorithm verifies that all star counts are non-negative, as negative values would indicate corrupted or invalid data.

Subsequently, the list of star counts is sorted in non-increasing order. The algorithm then iterates over the sorted list while maintaining an index representing the current candidate value of the h-index. For each position i, the algorithm checks whether the number of stars of the corresponding repository is greater than or equal to i. If this condition holds, the candidate h-index is updated. The iteration stops as soon as the condition is no longer satisfied.

The final value obtained in this process represents the GitHub h-index of the user. The time complexity of the algorithm is dominated by the sorting step and is therefore $O(n \log n)$, where n is the number of repositories.

The complete implementation of the described algorithm is provided in Algorithm 1.1, which appears below.

Algorithm 1.1: Python implementation of the GitHub h-index

```
def calculate_h_index_from_stars(star_counts):
    sorted_stars = sorted(star_counts, reverse=True)
    h_index = 0
    for i, stars in enumerate(sorted_stars, start=1):
        if stars >= i:
            h_index = i
        else:
            break
    return h_index
```

1.1.4 Stars as Citations: Why Not Forks?

We chose stars instead of forks. What are citations in scientific work? They are acknowledgment of the original work, showing the importance of the original work. Stars, in turn, show the importance of the repository, if some people saved it for themselves, it means that they recognized it in something, as well as scientific articles. Forks, on the other hand, are creating a new version of the existing work, as an extension of the already existing work, creating something new not necessarily having a connection with the original idea. Therefore, stars are much closer to citations than forks.

It is also worth mention that people add stars more often than they add fork. Therefore this also brings stars closer to citations. Creating new work based on old one is much harder than put star on it, as you can see on Figure 1.2.

Figure 1.2: Comparison of the number of stars and forks on GitHub.

1.1.5 Limitations of the GitHub h-index

Repositories and scientific papers are not the same. An important difference between them is that repositories unlike scientific articles need to be maintained. That is a scientific work is published once into the world. At the same time a repository on GitHub needs to be maintained to meet the realities of new technologies and updates.

Stars are not citations. To cite an article a scientist needs to make his own article. While to add a star to a repository on GitHub just needs to click one time. But it is not just a click. Mainly on GitHub users use stars as bookmarks. As you can see on Figure 1.3.

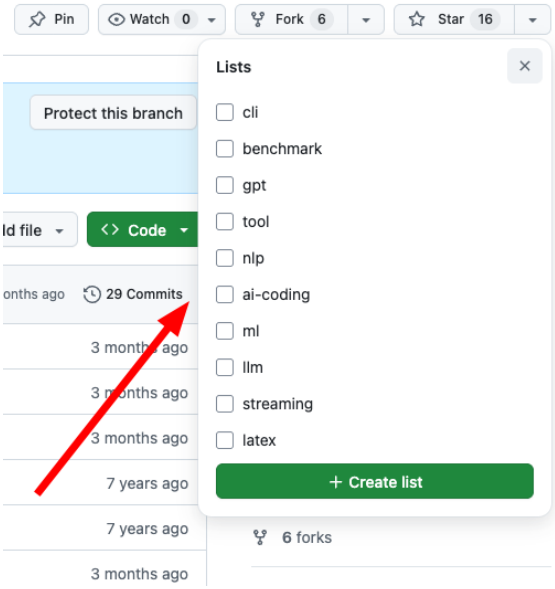

Figure 1.3: Stars used as bookmarks on GitHub repositories.

Other limitations of h-index on GitHub are that unlike citations stars can be taken back. That is stars on a repository can become less. However in the context of h-index this should not have such a big impact on users with very popular repositories. But if a user has an h-index and 10 repositories with about 10 stars then stars taken back from several repositories can very strongly change their h-index. For example if 4 stars leave from 4 repositories then the users h-index will become 6.

Another problem of stars on GitHub is that they can be bought. Different variants with exchanging stars. Buying stars and receiving them through bots. This is one of the problems. There are existing attempts to solve this problem, however all of them are very approximate and inaccurate, as it is very difficult to separate a bot account

from a normal user. It is also very difficult to determine users who have exchanged stars with each other from regular users.

Chapter 2

Data Collection from GitHub

This chapter describes the process of collecting data from GitHub required for calculating the GitHub h-index for a large number of users. The main challenge consisted in enumerating a comprehensive set of public GitHub users while adhering to API limitations.

2.1 GitHub API Rate Limits per User Account

GitHub enforces rate limits on both its REST [6] and GraphQL APIs [5] to prevent abuse and ensure fair access and service availability. These primary rate limits apply per authenticating entity—specifically, per user for requests authenticated via personal access tokens or on behalf of the same user (e.g., through OAuth apps).

2.1.1 REST API Primary Rate Limits per User

The REST API enforces a request-based rate limiting system.

For authenticated requests:

- Standard limit per user: 5,000 requests per hour.
- This quota is shared across all authentication methods acting on behalf of the same user (personal access tokens, OAuth apps, or GitHub Apps using user access tokens).

For unauthenticated requests:

- Limit: 60 requests per hour, tied to the originating IP address rather than a user account.

Rate limit status can be monitored via response headers such as X-RateLimit-Limit, X-RateLimit-Remaining, and X-RateLimit-Reset.

2.1.2 GraphQL API Primary Rate Limits per User

The GraphQL API uses a point-based system to better account for query complexity, with each query costing a minimum of 1 point, calculated based on the estimated number of underlying REST-equivalent requests, roughly total paginated requests divided by 100 and rounded up.

For authenticated requests:

- Standard limit per user: 5,000 points per hour.
- This quota is shared across all authentication methods acting on behalf of the same user.

Rate limit status is reported via similar headers or directly queryable through the rateLimit field in GraphQL responses.

2.1.3 Rationale for Using the GraphQL API over the REST API

Rate limit protocols are employed in REST APIs. These limits are generally based on the number of requests made within a specific timeframe. This approach is simpler to track but does not always account for the fact that some requests are significantly heavier than others.

In contrast, a system of resource and query governance is used in GraphQL APIs, which is arguably more precise than that of REST APIs. In REST APIs, data collection from three different resources requires separate requests, each of which counts against the rate limit. In GraphQL APIs, multiple REST calls can often be replaced by a single query. This means that, although the GraphQL "limit" might appear more restrictive due to complexity scoring, more relevant data can be fetched with a single unit of limit consumption than is typically possible with one REST request.

In summary, although both APIs impose comparable hourly quotas, the point-based system of the GraphQL API more accurately reflects actual resource consumption. This allows complex queries to retrieve equivalent data with fewer requests compared to the REST API. Such efficiency renders GraphQL particularly advantageous for large-scale data collection tasks involving interconnected resources. Consequently, the GraphQL API is prioritized in the subsequent implementation of this work to maximize data yield within the given rate limit constraints.

2.1.4 Multi-Token Rotation

To address the fact that each token has such a small number of limits per hour, it was decided to create token rotation by creating a `TokenManager` class. Several tokens

are loaded from the environment configuration during initialization. Then the manager distributes requests using all available tokens. This solution made it possible to increase the data parsing speed by 8 times.

Each token is monitored for rate limit status. If a token encounters an HTTP 403 error, this means that it has reached its request limit for that hour, and other tokens are tried in its place. When tokens run out, the program attempts to use them at intervals. Then, when it becomes possible to use a token, parsing continues.

2.2 Collecting GitHub Users

The first step in our analysis was to gather a dataset of GitHub users for whom the h-index would be calculated. The goal was to collect as many relevant users as possible; however, conventional scraping methods were not sufficient due to several limitations.

2.2.1 Challenges in User Collection

One major limitation of the GitHub API is the restriction to a maximum of 1,000 users per query. This limit arises because the API returns at most 100 users per page and allows a maximum of 10 pages per query—that is, 100 users per page multiplied by 10 pages equals 1,000 users. Consequently, multiple queries were required to retrieve a substantial portion of the user base.

An alternative approach considered was to filter users by their unique IDs and iterate sequentially through the entire GitHub population. However, preliminary estimates indicated that this method would be extremely time-consuming and resource-intensive. Moreover, the majority of GitHub accounts exhibit very low activity or negligible h-index values; thus, collecting data on millions of users with an h-index of 0 or 1 would contribute little value.

2.2.2 Focusing on Popular Users

To address these limitations, users with the highest number of followers were prioritized, under the assumption that such users are more likely to exhibit significant activity and a meaningful h-index [1]. Despite this approach, the GraphQL API's restriction of a maximum of 1,000 users per query remained a challenge.

Initially, this was mitigated by using the follower count of the last user in each retrieved batch as a threshold for the subsequent query. However, this method proved insufficient, as more than 1,000 users often share the same follower count, leading to the potential omission of relevant users.

To improve the filtering process, an additional parameter—the user's registration date—was introduced alongside the follower count. By combining these two criteria, the user space could be effectively partitioned into manageable segments, enabling the complete retrieval of relevant users within each segment.

2.2.3 Checkpointing and Resume Capability

Due to the need for long-term user parsing, the possibility that the parser might stop working was taken into account. Therefore, a checkpoint system was created. It records which batch of users was parsed. In case of a restart, the parser starts from the last recorded point. This allows the parser to work correctly and not start from the beginning in case of an internet failure or other problems.

2.2.4 Results

Using this strategy, data collection ran continuously for about one week and ultimately yielded information on approximately 3.5 million users.

Batches containing users with a large number of followers were processed completely, while batches of users with fewer followers were only partially collected, limited to the maximum of 1,000 users per segment query. This partial collection occurred because, in segments with low follower counts—particularly those with 0 followers—far more than 1,000 users could share the exact same follower count and fall within a narrow registration date range, exceeding the API's per-query limit. Fully parsing these denser, low-activity segments would have required further subdivision, but was not pursued due to the diminishing returns: these batches mainly consist of less active users with lower h-index values, so their partial omission has only a minor effect on the overall quality and representativeness of the dataset.

2.3 Collecting Repository Data and Computing the GitHub h-index

Once the set of approximately 3.5 million GitHub users had been assembled, the next phase involved retrieving data on their repositories to enable the calculation of the GitHub h-index.

2.3.1 Challenges in Repository Retrieval

Repository data for each user was fetched using the GraphQL API, as it allowed efficient retrieval of repository lists along with essential fields such as the stargazer count.

Queries were structured to paginate through a user's repositories and extract the number of stargazers for each. Initial implementations processed users sequentially using a single personal access token. This approach proved to be extremely slow, achieving a throughput of approximately 10,000 users per day. Given the dataset size of 3.5 million users, completion would have required an estimated 350 days—an impractically long duration that also risked interruptions due to rate limit resets or network issues. The primary bottleneck was identified as the sequential nature of the processing, combined with the time required for API responses and pagination through repositories of highly active users.

2.3.2 Implementation of Parallel Processing

To collect data faster, we used parallel processing. Because each token has a limited number of requests per hour, we added token rotation as described in Section 2.1.4. Several tokens are loaded from the environment configuration at startup. The manager spreads requests across all available tokens, which increases the hourly limit in practice. We processed users in parallel in batches of five, and each batch was assigned to a separate token and run in parallel threads.

We track the rate limit status for each token. If a token gets an HTTP 403 error, it means that token has reached its request limit for the hour, so we switch to other tokens. When all tokens are used up, the program tries them again after short waits. When a token becomes available again, parsing continues. This approach increased data parsing speed by 8 times. As a result, repository data collection for 3.5 million users finished in about five days. We also monitored rate limit headers during the whole process to keep token usage balanced and add short pauses when needed, so hourly limits were not exhausted.

2.3.3 Results and Efficiency Considerations

The parallelized approach not only reduced the collection time dramatically but also ensured that the dataset remained fresh, minimizing the impact of ongoing changes in repository stargazer counts. The use of multiple tokens and careful rate limit management proved essential for scaling the operation to millions of users while adhering to GitHub's API policies. This phase concluded the raw data acquisition, paving the way for subsequent analysis of the GitHub h-index distribution across the collected user base.

2.4 Profile Enrichment and Metadata Collection

After the initial user enumeration phase, the collected user records contained only basic identifiers. To enable meaningful feature engineering for the machine learning models described in Chapter 3, additional profile metadata was required. This section describes the implementation of the profile enrichment pipeline, which fetches extended user information via the GitHub GraphQL API.

2.4.1 Metadata Extraction

It was decided to extract additional data for each user. The corresponding mapping of collected metadata is shown in Table 2.1.

Table 2.1: Database Mapping of Collected Metadata

Database Column	Description
Network Metrics	
followers_count	Number of followers
following_count	Number of following users
organizations_count	Number of organization memberships
Profile and Content	
bio	Biography text content
bio_length	Length of biography string
company	Employer or organization name
location	Geographic location

In this stage, we used the previously described parallel processing approach Section 2.3.2 and checkpointing mechanism Section 2.2.3.

2.4.2 Social Link Extraction and Normalization

Links were collected in several ways. Extract links from GitHub fields directly through GraphQL: the fields websiteUrl, twitterUsername, and email. Obtain links using regex from bui, company, and location collected during metadata gathering. If the amount of link data obtained was too small, then download the user's full HTML and use regex to search for links in it.

After obtaining all links, they were divided into platforms: Twitter/X, LinkedIn, Instagram, YouTube, Mastadon, and Telegram. There was also a priority system so that if a link could belong to several types at the same time, it was first assigned

according to the selected priority. Thus, telegram has priority 1, and website has priority 99. A function was created that receives a link and determines its type.

2.5 Data Storage Architecture

For data storage, a lightweight SQLite database was used. SQLite was selected due to its simplicity, serverless architecture, zero-configuration requirements, and sufficient performance for the scale of this application. Although the data collection pipeline utilizes parallel processing to maximize throughput, the system does not require concurrent access by multiple external processes or users, making SQLite's single-file database approach ideal. Additionally, direct use of SQL statements (via Python's built-in `sqlite3` module) was preferred over higher-level object-relational mappers (ORMs) such as SQLAlchemy or Prisma. This decision was made to maintain full control over the schema, ensure predictable query performance, avoid unnecessary abstraction overhead in a relatively simple data model, and keep external dependencies minimal.

The database consists of three tables: `users`, `repositories` and `user_links`. The schema, including constraints and indexes, is defined by the following SQL statements:

Algorithm 2.1: SQL schema definition for the SQLite database

```
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    github_id BIGINT NOT NULL UNIQUE,
    github_name TEXT NOT NULL UNIQUE,
    created_at TEXT NOT NULL,
    last_updated TEXT NOT NULL,
    links_parsed_at TEXT,
    h_index INTEGER NOT NULL DEFAULT 0,
    repositories_fetched BOOLEAN NOT NULL DEFAULT 0,
    followers_count INTEGER NOT NULL DEFAULT 0,
    following_count INTEGER NOT NULL DEFAULT 0,
    sponsors_count INTEGER NOT NULL DEFAULT 0,
    organizations_count INTEGER NOT NULL DEFAULT 0,
    company TEXT,
    location TEXT,
    bio TEXT,
    bio_length INTEGER NOT NULL DEFAULT 0
);

CREATE TABLE IF NOT EXISTS repositories (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
    user_id INTEGER NOT NULL,
    repository_id BIGINT NOT NULL,
    name TEXT NOT NULL,
    created_at TEXT NOT NULL,
    stars INTEGER NOT NULL CHECK (stars >= 0),
    FOREIGN KEY (user_id) REFERENCES users (id) ON DELETE
       CASCADE,
    UNIQUE (user_id, repository_id)
);

CREATE TABLE IF NOT EXISTS user_links (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    url TEXT NOT NULL,
    link_type TEXT DEFAULT 'unknown',
    created_at TEXT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users (id) ON DELETE
       CASCADE,
    UNIQUE (user_id, url)
);

CREATE INDEX IF NOT EXISTS idx_user_github_id ON users(github_id
   );
CREATE INDEX IF NOT EXISTS idx_user_github_name ON users(
   github_name);
CREATE INDEX IF NOT EXISTS idx_users_need_fetch ON users(
   repositories_fetched);
CREATE INDEX IF NOT EXISTS idx_users_followers ON users(
   followers_count);
CREATE INDEX IF NOT EXISTS idx_users_h_index ON users(h_index);
CREATE INDEX IF NOT EXISTS idx_users_links_parsed ON users(
   links_parsed_at);
CREATE INDEX IF NOT EXISTS idx_repo_user ON repositories(user_id
   );
CREATE INDEX IF NOT EXISTS idx_user_links_user_id ON user_links(
   user_id);
CREATE INDEX IF NOT EXISTS idx_user_links_url ON user_links(url)
   ;
```

The overall structure of the database, including the relationship between the three tables, is illustrated in Figure 2.1.

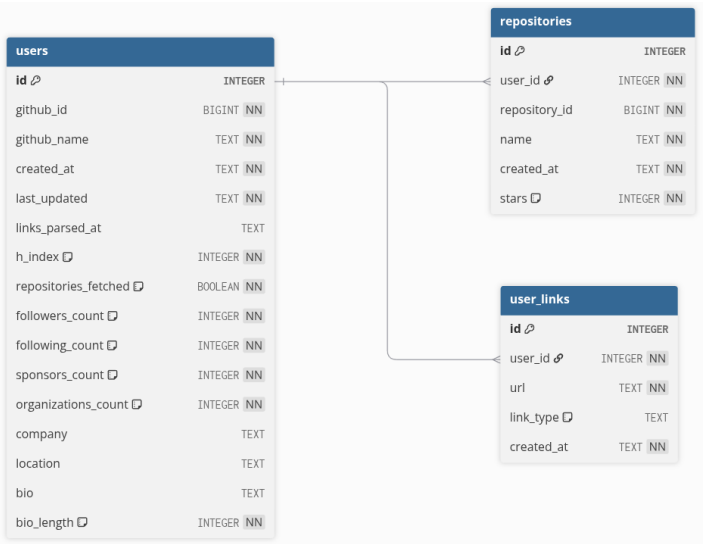

Figure 2.1: Entity-Relationship Diagram (ERD) of the database schema

Chapter 3

Machine Learning Models for GitHub h-index Prediction

3.1 Model Architecture

The Random Forest algorithm is an ensemble learning method that operates by constructing a multitude of decision trees during training and outputting the mean prediction of the individual trees for regression tasks [2]. Each tree is built using a bootstrap sample of the training data, and at each split node, only a random subset of features is considered. This dual randomness reduces the variance of the model and mitigates overfitting compared to a single decision tree.

Random Forest was selected for this thesis because it allows working with the top 10% of users during model training, which helps predict extreme values. It is more robust to outliers than classical regression approaches, making robustness the primary reason for this choice. Additionally, Random Forest provides native feature importance metrics, which facilitate the analysis of which GitHub metrics contribute most to the h-index prediction.

3.2 What the Model Predicts

The model predicts the estimated h-index value for a GitHub user.

3.2.1 Prediction Output Interpretation

To make it easier to assess a developer's impact, predicted h-index values are grouped into four levels, as shown in Table 3.1.

Table 3.1: Interpretation of predicted h-index values

Output Value	**Interpretation**	**Practical Meaning**
0–2	Low impact	Low impact developer
3–10	Moderate impact	Consistent contributor
10–30	High impact	Recognized developer
30+	Very high impact	Influential maintainer

The model output has some error. As shown in Section 3.4, the mean absolute error is low for most users but gets higher for very large values with h-index above 30. So the model works well for broad impact levels such as low and high, but predicted values for top users should be treated as not exact counts.

For the log2-transformed model, the raw output is inverse-transformed using Equation 3.7 before interpretation, ensuring the value aligns with the original h-index scale described in Table 3.1.

3.3 Training Process

Model training consists of 4 steps: data preparation, feature creation, configuration setup, and model evaluation.

3.3.1 Data Preparation and Splitting

The dataset was split into training and test subsets using an 80/20 distribution, with a fixed seed so that this model can be trained again. The `train_test_split` function from the `sklearn` library was used for splitting. Before the start of training, several stages were completed:

1. Data cleaning, during the collection of the dataset itself it was done so that there was no empty h-index, but for greater protection this step was added.

2. Numeric features with an absolute skewness coefficient exceeding 0.8 underwent a $\log(1+x)$ transformation to stabilize variance and enhance model convergence. However, variables such as `followers_count`, `following_count`, `organizations_count`, `bio_length`, and `repo_count` were deliberately left untransformed to preserve their interpretability.

3.3.2 Feature Engineering

For a better understanding of the model, several additional features were created, which show the ratio of other features to each other.

$$\text{followers\_per\_repo} = \frac{\text{followers\_count}}{\text{repo\_count} + 1} \tag{3.1}$$

$$\text{following\_to\_followers} = \frac{\text{following\_count}}{\text{followers\_count} + 1} \tag{3.2}$$

$$\text{stars\_per\_repo} = \frac{\text{total\_stars}}{\text{repo\_count} + 1} \tag{3.3}$$

The complete set of 13 features used for training is summarized in Table 3.6.

3.3.3 Model Configuration

Two Random Forest models were trained: one trained on h-index, the second on log2-transformed h-index. Both models were trained and described for a better understanding of the formation of h-index from features.

The models were trained using RandomForestRegressor from the sklearn library. Training was parallelized across all available CPU cores to reduce computation time. Then the trained models were saved using joblib for further use and analysis without repeated training.

3.3.4 Training Execution

Both models took approximately the same time during training, about 2.5 minutes. Both models were trained on the same training subset and evaluated on the same subset as well.

3.4 Performance Evaluation

3.4.1 Evaluation Framework & Metrics Definition

MAE and the R^2 coefficient were selected to evaluate the models. MAE is robust to outliers, and R^2 enables comparison across models.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{3.4}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{3.5}$$

Evaluation used an 80/20 train/test split, with a fixed random seed to prevent leakage.

3.4.2 Quantitative Results

Table 3.2 summarizes the model's performance on the test set, reported both for the full population and for the top 10% of users by h-index. This stratification allows us to assess whether the model maintains accuracy for high-impact developers, who are often of primary interest in talent scouting and research impact estimation.

Table 3.2: Performance of baseline Random Forest model (raw h-index target)

Metric	**Overall**	**Top 10% ($h \geq 3.0$)**
MAE	0.150	0.712
R^2	0.9241	0.8172
Sample size	734,299	87,373 (11.9%)

Table 3.3 presents the performance of the Random Forest model trained on the log2-transformed target variable in Equation 3.6. Predictions were inverse-transformed back to the original h-index scale before evaluation, ensuring direct comparability with the baseline model. As before, results are stratified by the full test population and the top 10% of users.

Table 3.3: Performance of Random Forest model with log2-transformed target

Metric	**Overall**	**Top 10% ($h \geq 3.0$)**
MAE	0.153	0.755
R^2	0.9142	0.7863
Sample size	734,299	87,373 (11.9%)

3.4.3 Effect of Target Variable Transformation

Because the dataset has a heavy-tailed distribution of the h-index, a $\log_2$ transformation was applied to h-index.

$$y' = \log_2(1 + y) \tag{3.6}$$

By using this transformation, the dynamic range of the h-index was compressed from $[0, 253]$ to $[0, 7.983]$, reducing the leverage of extreme outliers during model training [10].

$$\hat{y} = 2^{\hat{y}'} - 1 \tag{3.7}$$

3.4.4 Error Analysis

As you can see in the figures for the classic model and the log2 model (Figures 3.1 and 3.2), the errors are distributed mostly for top cases, but in most situations the error is only 1–4 in h-index. But there are situations when it has a big difference, and those are situations when the h-index is about 30.

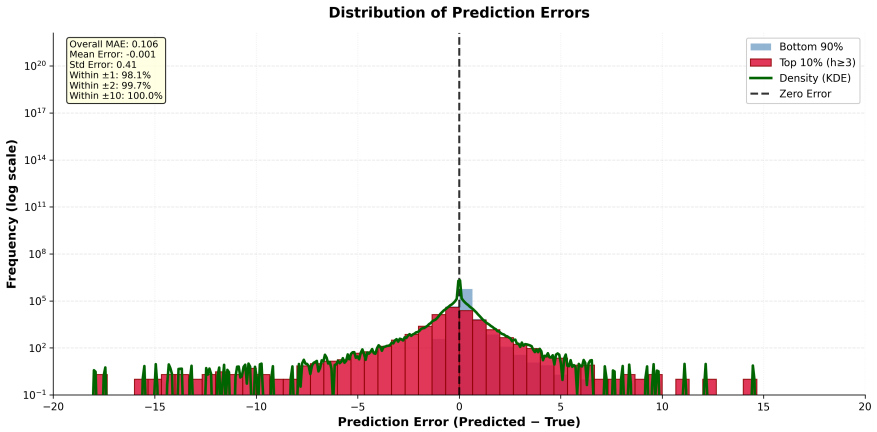

Figure 3.1: Distribution of prediction errors ($\hat{y} - y$) for the baseline model.

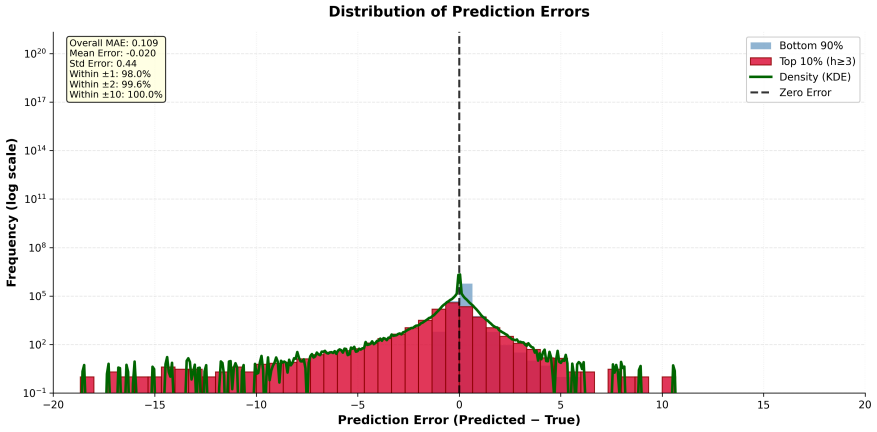

Figure 3.2: Distribution of prediction errors ($\hat{y} - y$) for the log2-transformed model.

For both models, errors are mostly distributed symmetrically. But for very big h-index, the distribution becomes more deflected to the left side.

3.4.5 Case Studies

To better understand how the model works, representative cases were extracted from the test set. Focusing on all user distribution, from high to low h-index.

Both models tables are shown in Tables 3.4 and 3.5. Both of them always under-predict extreme h-index values, users A–D, indicating difficulty extrapolating beyond

Table 3.4: Case studies: Predicted vs. true h-index for high-impact users

User	True h	Pred. h	Error	Profile Characteristics
A	253	63.7	-189.3	1118 repos, 77,090 followers
B	151	58.4	-92.6	874 repos, 23,354 followers
C	136	71.2	-64.8	299 repos, 75,482 followers, no organizations
D	80	52.0	-28.0	367 repos, 20,493 followers
E	69	66.6	-2.4	170 repos, 36,335 followers, no organizations
F	50	25.7	-24.3	62 repos, 166 followers, no organizations
G	1	1.0	+0.0	21 repos, 3 followers, no organizations

Table 3.5: Case studies log2: Predicted vs. true h-index for high-impact users

User	True h	Pred. h	Error	Profile Characteristics
A	253	55.7	-197.3	1118 repos, 77,090 followers
B	151	52.3	-98.7	874 repos, 23,354 followers
C	136	60.3	-75.7	299 repos, 75,482 followers, no organizations
D	80	38.7	-41.3	367 repos, 20,493 followers
E	80	49.5	-30.5	268 repos, 6,802 followers, high following_count (137)
F	69	57.9	-11.1	170 repos, 36,335 followers, no organizations
G	1	1.0	+0.0	21 repos, 3 followers, no organizations

the training distribution.
The log2-transformed model Table 3.5 shows slightly under-prediction for the highest h-index users.
Users with moderate h-index and balanced feature profiles are predicted accurately.
Near-zero h-index is mostly predicted correctly. It can be because models know *total_stars*. That very helps the model to understand in low h-index cases. Even with 100 repositories, user can have 100 stars, which makes prediction in mostly case obviously.
High followers count do not guarantee hight h-index prediction. Features like *stars*, *repo_count* dominate there.

3.5 Features

We selected 13 features that capture three complementary dimensions of GitHub user impact: repository engagement, social network signals, and profile metadata.

Highly skewed numeric features were log-transformed using $\log(1 + x)$ to stabilize variance and improve model convergence. Derived ratio features normalize absolute counts by repository volume, helping distinguish between prolific but low-impact users and focused high-impact contributors.

Table 3.6: Summary of features used for h-index prediction

Feature	**Transform**	**Interpretation**
`repo_count`	None	Repository volume
`total_stars`	$\log(1 + x)$	Cumulative code impact
`max_stars`	$\log(1 + x)$	Peak impact repository
`star_variance`	$\log(1 + x)$	Consistency of impact
`stars_per_repo`	$\log(1 + x)$	Normalized engagement
`followers_count`	None	Social reach
`following_count`	None	Community participation
`followers_per_repo`	$\log(1 + x)$	Reach per project
`following_to_followers`	$\log(1 + x)$	Reciprocity ratio
`organizations_count`	None	Institutional affiliation
`bio_length`	$\log(1 + x)$	Profile completeness

Below we detail the interpretation for each feature:

`repo_count` Total number of public repositories owned by the user.

`total_stars` Sum of stars across all user repositories.

`median_stars` Median stars per repository.

`max_stars` Stars on the user's most popular repository.

`star_variance` Variance in stars across repositories.

`stars_per_repo` `total_stars` / (`repo_count` + 1).

`followers_count` Number of users following this account.

`following_count` Number of accounts this user follows.

`followers_per_repo` `followers_count` / (`repo_count` + 1).

`following_to_followers` `following_count` / (`followers_count` + 1).

`organizations_count` Number of GitHub organizations the user belongs to.

`bio_length` Character count of the user's profile biography.

3.6 Feature Importance Analysis

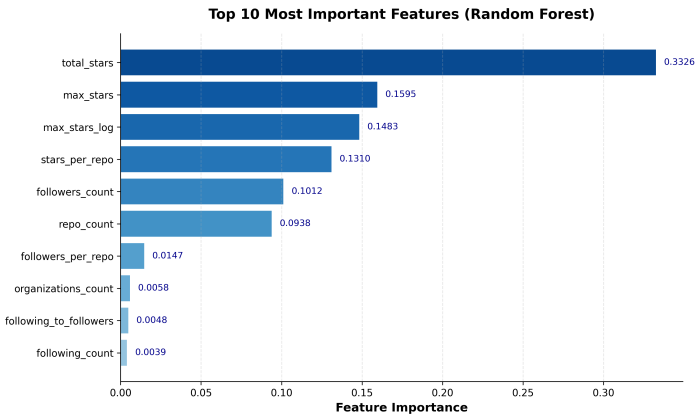

Figure 3.3: Feature Importance Distribution for Random Forest Model

In both Figures 3.3 and 3.4, the most important metric is `total_stars`. Figure 3.4 log2 model places more importance on repository-level metrics such as `stars_per_repo`. Compared with the classic model in Figure 3.3, the log2 model in Figure 3.4 relies less on `repo_count` because it captures similar information through `stars_per_repo`.

The number of followers has a low impact on the user h-index. `followers_count` and `followers_per_repo` have low importance. Features such as `bio_length`, `organizations_count`, and `following_to_followers` have negligible importance and are not meaningful predictors in the model.

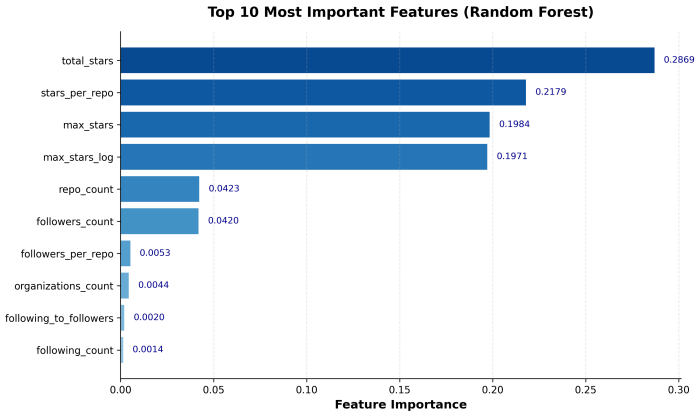

Figure 3.4: Feature Importance Distribution for Log2 Random Forest Model

3.7 Comparative Discussion

Comparison of two models, the standard one and the log2-transformed model.

3.7.1 Model Performance Comparison

As can be seen in the error graphs Figures 3.1 and 3.2, the classical model predicts h-index better than the log2-transformed model. However, both models still face a problem when they need to predict high h-index values.

3.7.2 Best Use Cases

Based on the obtained results, the models are best used in these situations:

- Baseline Model: Recommended for applications requiring precise estimation of h-index for the majority of users h < 30. Since the majority of the dataset falls within this range, the baseline model's lower overall MAE makes it more suitable for general talent scouting or ranking systems where exact integer values matter.
- Log2-Transformed Model: Preferable when the goal is to distinguish between orders of magnitude of impact rather than exact values. For example, distinguishing a novice contributor h ≈ 1 from an influential maintainer h ≈ 100 is more robust in log-space. This model is also theoretically safer for deployment if the incoming data distribution shifts to include even more extreme outliers than seen in the training set, as it is less sensitive to the scale of the target variable.

Chapter 4

Empirical Analysis of User Behavior

4.1 Social Network Presence

Figure 4.1 illustrates the distribution of external social media links found within the analyzed GitHub user profiles. To focus on community and networking behaviors, personal websites and email addresses were excluded from this specific visualization. The data represents a subset of 812,429 users who have linked at least one social platform to their profile.

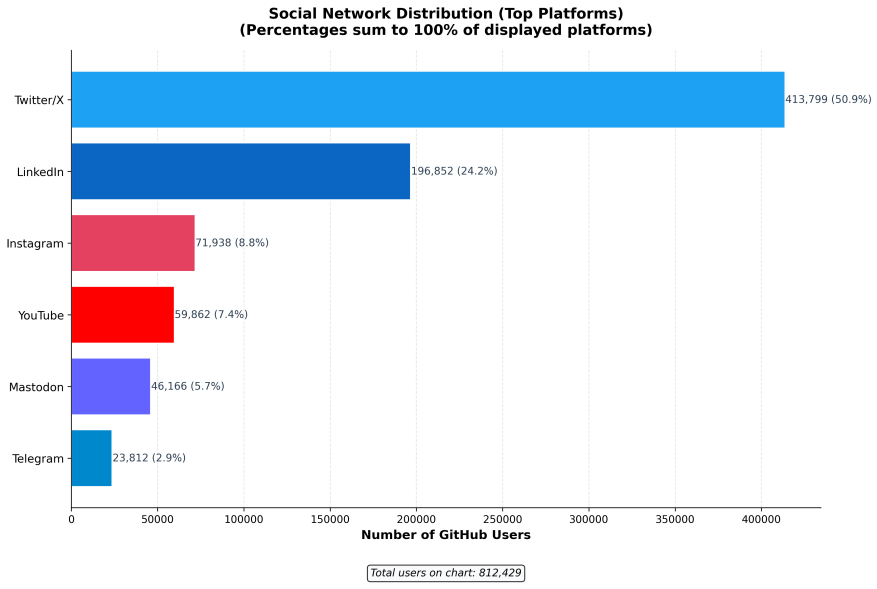

Figure 4.1: Prevalence of External Social Network Links in GitHub User Profiles (Excluding Personal Websites)

The most popular social network turned out to be Twitter. Second place was taken by LinkedIn. But Twitter turned out to be twice as popular, occupying 50.9 percent. LinkedIn occupies approximately as many percent as all the remaining social networks.

Among GitHub users, the most popular social network is Twitter, but a large

number of users have LinkedIn.

4.1.1 Top h-index Users Analysis

It was decided to analyze who these people with the highest h-index on GitHub are and what they did to make it so high.

Following the approach of Hirsch [7], who analyzed the characteristics of scientists with the highest h-indices, including multiple Nobel laureates, we examine the profiles of GitHub users with the highest h_G values in our dataset. This analysis helps explain the metric and what leads to the highest impact on the platform.

Table 4.1: Top developers by h-index on GitHub

Rank	Developer	h-index	Followers	Total Stars	Repositories
1	sindresorhus	253	77,090	986,598	1118
2	keijiro	151	23,354	102,585	874
3	lucidrains	137	58,746	177,669	310
4	bradtraversy	136	75,482	179,625	299
5	llSourcell	113	36,805	68,754	373
6	egoist	102	13,210	77,099	737
7	antfu	102	38,264	62,495	244
8	tj	99	51,516	140,790	219
9	adrianhajdin	97	36,153	121,540	143
10	geerlingguy	90	26,172	69,630	271
11	mafintosh	89	6,323	50,437	665
12	mattn	85	13,182	56,774	1065
13	arpitbbhayani	84	7,139	17,235	155
14	WebDevSimplified	82	32,164	26,844	220
15	Hrishikesh332	82	143	9,695	116
16	krishnaik06	81	40,190	51,373	339
17	phodal	80	20,493	88,162	367
18	ankane	80	6,802	84,452	268
19	lukeed	79	4,996	57,985	214
20	iamshaunjp	79	32,276	27,907	146

Hirsch found that Nobel Prize winners in physics usually have the highest h-index scores. We found the same thing on GitHub, the people who have the highest GitHub

scores. However, there is an important difference. In academia, citations take decades to build up. On GitHub, stars can grow very fast if a project becomes popular or trends online. Because of this speed, the GitHub h-index reflects recent success much quicker than the academic version does.

4.1.2 Case Studies: Top 5 GitHub Users by h-index

The following subsections present detailed profiles of the five GitHub users with the highest h-index on GitHub.

1. Sindre Sorhus (`sindresorhus`)

Sindre Sorhus, mostly known as an npm package maintainer, created more than a thousand repos and made many small tools to solve local problems, mostly on macOS. He also made an awesome repository, a collection of different programming things. You can see his GitHub profile in Figure 4.2.

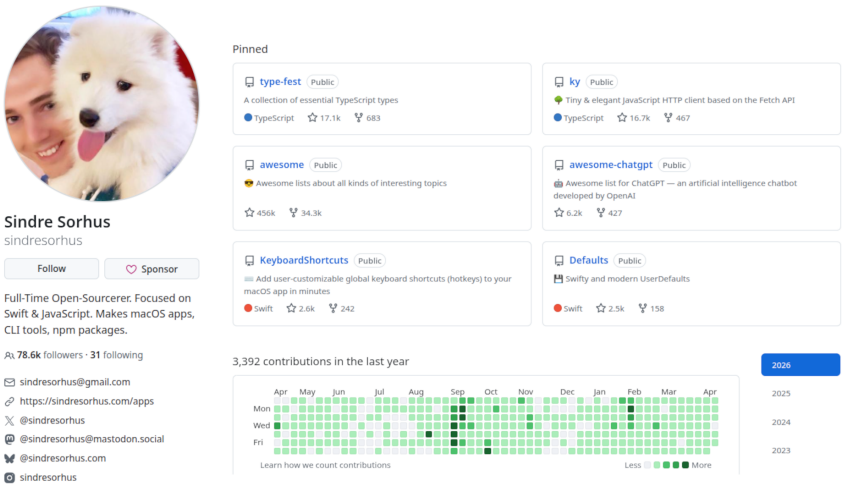

Figure 4.2: Sindre Sorhus GitHub Profile ($h_{index} = 253$)

2. Keijiro Takahashi (`keijiro`)

Keijiro Takahashi, mostly known as specialized Unit-related developer. He is from Japan. Works in Unity Technologies Japan. His most famous repositories are AICommand, ChatGPT integration with Unity Editor. AIShader, ChatGPT-powered shader generator. And different custom effects for Unity. You can see his GitHub profile in Figure 4.3.

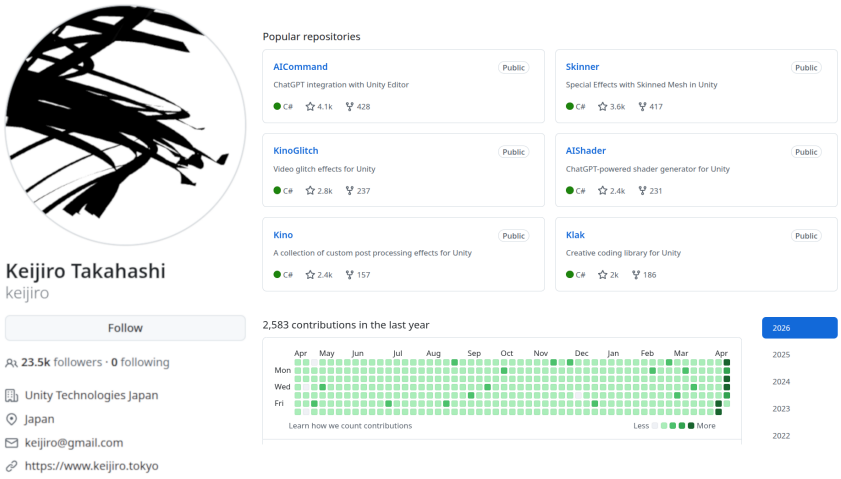

Figure 4.3: Keijiro Takahashi GitHub Profile ($h_{index} = 151$)

3. Phil Wang (`lucidrains`)

Phil Wang is a developer specializing in PyTorch with implementing deepl learning models. His works are mainly in transformer architecture and attention mechanism. His most pupular repositories are Vission Transformers, DALL-E 2. And other transformed-based models and text-to-image generation. You can see his GitHub profile in Figure 4.4.

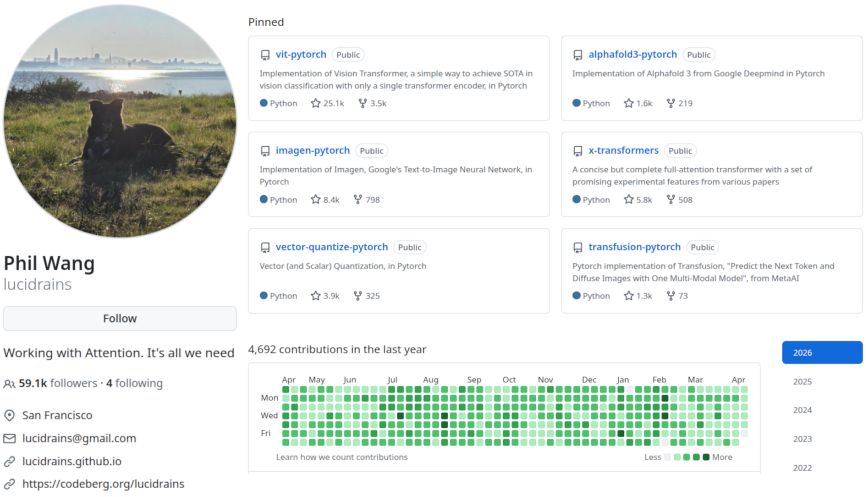

Figure 4.4: Phil Wang GitHub Profile ($h_{index} = 137$)

4. Brad Traversy (`bradtraversy`)

Brad Traversy is a full stack web developer and online instructor, best known for his Traversy Media YouTube channel. His GitHub project motly have some education things, like 50projects50days or some collection like design-resources-for-developers.

His repositories focus on practical learning examples. You can see his GitHub profile in Figure 4.5.

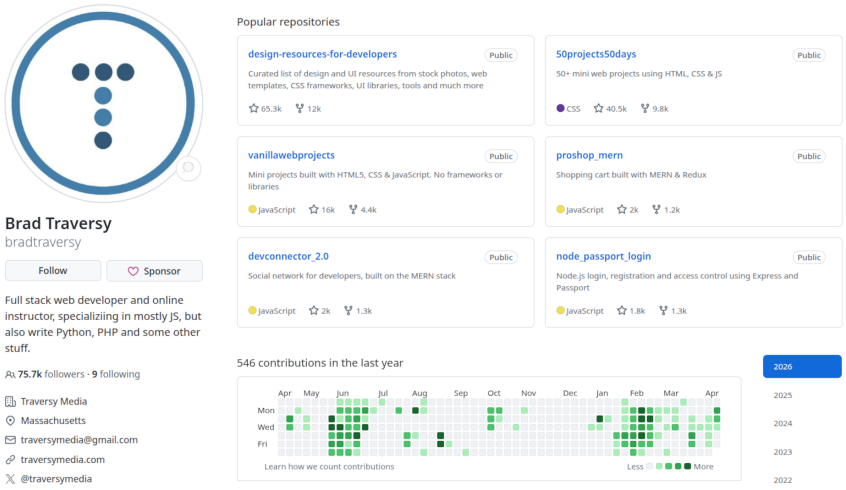

Figure 4.5: Brad Traversy GitHub Profile ($h_{index} = 136$)

5. Siraj Raval (`llSourcell`)

Siraj Raval is a Senior AI/ML Engineer and popular YouTube educator specializing in machine learning. His GitHub repositories contain educational code along with his YouTube videos, including popular series like Learn Machine Learning in 3 Months. His content focuses on making complex ML topics accessible to beginners through practical projects and tutorials. You can see his GitHub profile in Figure 4.6.

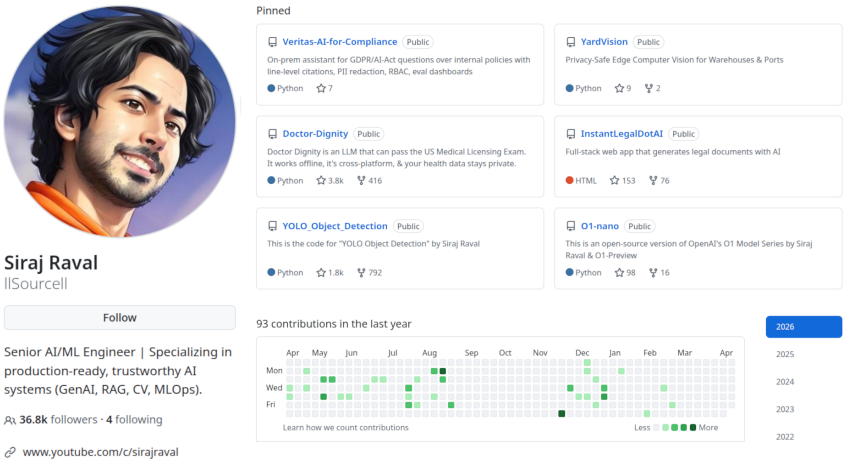

Figure 4.6: Siraj Raval GitHub Profile ($h_{index} = 113$)

4.2 Correlation Analysis of GitHub Metrics

The Spearman's rank correlation coefficient is calculated as follows [11]:

$$\rho = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n(n^2 - 1)} \tag{4.1}$$

where $d_i = \text{rank}(x_i) - \text{rank}(y_i)$ represents the difference between the ranks of the i-th observation in each variable, and n is the sample size.

To understand the relationships between different GitHub features, we computed Spearman's rank correlation coefficient. Unlike Pearson correlation, Spearman's correlation assesses monotonic relationships and does not assume linear relationships, making it more robust to outliers and suitable for analyzing metrics with non-normal distributions.

Figure 4.7 presents the correlation matrix of key GitHub features including h-index, number of followers, total stars, and repository count.

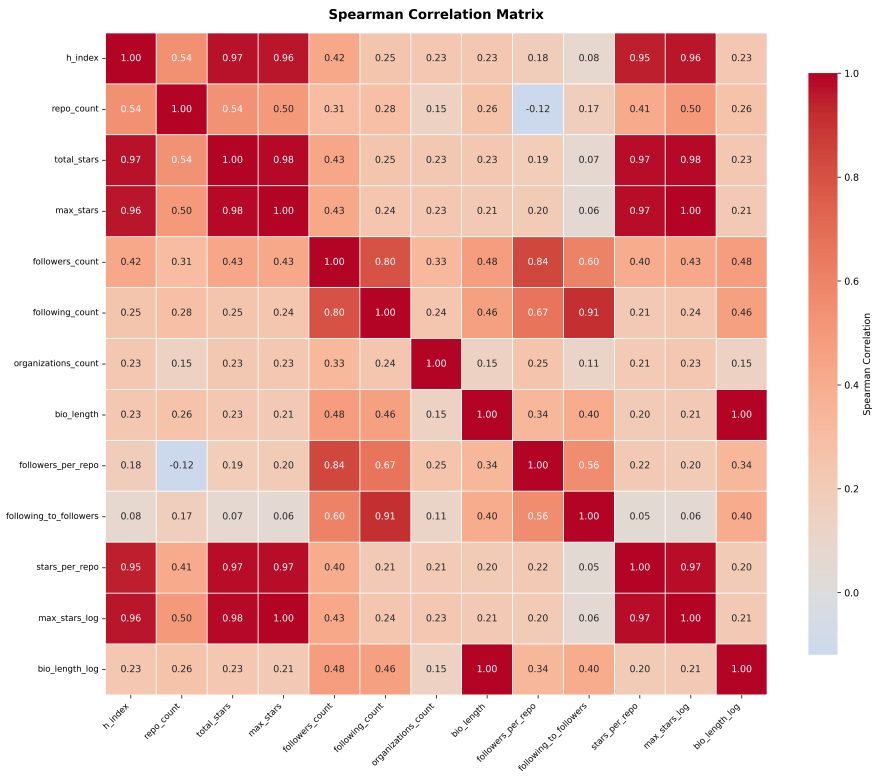

Figure 4.7: Spearman Correlation Matrix of GitHub Metrics

Correlation analysis revealed several important findings. It reaffirmed the importance of the total number of stars, and also indicated the importance of the maximum number of stars and the average number of stars in all repositories. It also confirmed a weak correlation between the number of subscribers and the h-index.

An interesting observation is the slight negative correlation between the number of repositories and the average number of subscribers across all of a user's repositories.

It is also worth noting that the potentially strong correlation between the total number of stars and the maximum number of stars suggests that users with a large number of stars typically receive most of them from a single repository.

Conclusions

We adapted the h-index for GitHub. We gathered information on 3.5 million users and about 54 million repositories. We analyzed all the collected data and found interesting points in it.

We found how the h-index is distributed and that it has very left-skewed data. We discovered that the h-index does not correlate well with the number of followers. We tested the assumption in practice that having many repositories does not mean a significant contribution. We found that the total number of stars is the most significant feature in predicting the h-index. We identified the developers with the highest h-index.

Limitations

- Stars can be bought or exchanged.
- Stars are used as the basis for developer evaluation, although this is not the only possible option.
- Private repositories were not taken into account.
- To get the date, you need to re-parse all repositories; the current results are fixed to the time when they were parsed.

Future Work

- Implement a system to avoid excluding less popular users and to parse everyone.
- Develop a system to detect bots and star farms.
- Add other metrics, such as forks and commits.
- Create separate models for analyzing the most popular users to reduce large deviations.

Bibliography

[1] Kelly Blincoe, Jigyasa Sheoran, Sean Goggins, Emil Petakovic, and Daniela Damian. Understanding the popular users: Following, affiliation influence and leadership on github. *Information and Software Technology*, 70:30–39, 2016.

[2] Leo Breiman. Random forests. Statistics Department, University of California, Berkeley, 2001.

[3] Emily Escamilla, Martin Klein, Talya Cooper, Vicky Rampin, Michele C. Weigle, and Michael L. Nelson. The rise of GitHub in scholarly publications. In G. Silvello, O. Corcho, P. Manghi, G.M. Di Nunzio, K. Golub, N. Ferro, and A. Poggi, editors, *Linking Theory and Practice of Digital Libraries*, volume 13541 of *Lecture Notes in Computer Science*, pages 187–200, Cham, 2022. Springer.

[4] Shawn P. Gilroy and Brent A. Kaplan. Furthering open science in behavior analysis: An introduction and tutorial for using github in research. *Perspectives on Behavior Science*, 42:565–581, 2019.

[5] GitHub. Rate limits and query limits for the graphql api. GitHub Docs, 2026.

[6] GitHub. Rate limits for the rest api. GitHub Docs, 2026.

[7] Jorge E. Hirsch. An index to quantify an individual's scientific research output. *Proceedings of the National Academy of Sciences*, 102(46):16569–16572, 2005.

[8] R. L. Larsen. The future of scholarly communication: Building the infrastructure for cyberscholarship. *Journal of the American Society for Information Science and Technology*, 58(7):1061–1066, 2007.

[9] Jeffrey Perkel. Democratic databases: science on github. *Nature*, 538:127–128, 2016.

[10] scikit-learn developers. Effect of transforming the targets in regression model. scikit-learn Documentation, 2026.

[11] Charles Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904.